Store Memory-Level Parallelism Optimizations for Commercial Applications

Yuan Chou, Lawrence Spracklen and Santosh G. Abraham
Advanced Processor Architecture
Scalable Systems Group
Sun Microsystems
{yuan.chou,lawrence.spracklen,santosh.abraham}@sun.com

Abstract

This paper studies the impact of off-chip store misses on processor performance for modern commercial applications. The performance impact of off-chip store misses is largely determined by the extent of their overlap with other off-chip cache misses. The epoch MLP model is used to explain and quantify how these overlaps are affected by various store handling optimizations and by the memory consistency model implemented by the processor. The extent of these overlaps are then translated to off-chip CPI. Experimental results show that store handling optimizations are crucial for mitigating the substantial performance impact of stores in commercial applications. While some previously proposed optimizations, such as store prefetching, are highly effective, they are unable to fully mitigate the performance impact of off-chip store misses and they also leave a performance gap between the stronger and weaker memory consistency models. New optimizations, such as the Store Miss Accelerator, an optimization of Hardware Scout and a new application of Speculative Lock Elision, are demonstrated to virtually eliminate the impact of off-chip store misses.

1 Introduction

Commercial applications such as online transaction processing (OLTP), web servers and application servers represent a crucial market segment for shared-memory multiprocessor servers. Numerous studies [1,2,3,4,5,6,7] have shown that many of these applications are characterized by large instruction and data footprints which cause high cache miss rates that result in high Cycles Per Instruction (CPI). While there has been substantial research to improve their instruction miss rates [1] as well as to mitigate the performance impact of their high load miss rates, there has been little research into the performance impact of stores in commercial applications. Table 1 shows that stores constitute a very significant percentage of the dynamic instruction count of commercial applications and store miss rates are comparable to, if not higher than, load miss rates and instruction miss rates. This raises the following questions: Do the stores in commercial applications stall the processor? And if so, what can be done to minimize or eliminate these stalls? It is these questions that this paper attempts to answer.

Off-chip misses (i.e. instruction and data accesses that miss the on-chip caches) require long latency accesses to either main memory or an off-chip cache and are particularly expensive. The performance impact of off-chip store misses depends on the degree of their overlap with computation and with other off-chip misses. Our previous paper [8] presented the epoch memory-level parallelism (MLP) model and used it to explain the impact of microarchitecture on the overlap of off-chip load misses and off-chip instruction misses. In this paper, the epoch MLP model is extended and used to explain and quantify the impact of store handling optimizations on the overlap between off-chip store misses and other off-chip misses.

Table 1: Store and miss rate statistics for 2MB 4-way set associative (64B line size) L2 cache.

Per 100 insts	Database	TPC-W	SPECjbb	SPECweb
Store frequency	10.09	7.28	7.52	7.20
L2 store miss rate	0.36	0.12	0.07	0.13
L2 load miss rate	0.57	0.06	0.25	0.14
L2 inst miss rate	0.09	0.06	0.00	0.01

The performance of stores is also impacted by the memory consistency model implemented by the processor. Previous detailed studies on memory consistency models were performed using scientific workloads rather than commercial workloads and most were focused on the performance differences between sequential consistency [9] and release consistency [10]. However, none of the four remaining server processor instruction set architectures implement the sequential consistency model. They either implement variations of processor consistency [11] (Intel/AMD x86 and x64, Sun SPARC TSO), weak consistency [12] (IBM PowerPC, Sun SPARC RMO) or release consistency (Intel IA-64). This paper contributes to the research of memory consistency models by using the epoch MLP model to explain and quantify how the processor consistency and weak consistency models affect the performance of stores in commercial applications.

The results in this paper show that the majority of off-chip store misses are not overlappable with computation. Store handling optimizations that improve store MLP, such as store prefetching, are illustrated to be critical in mitigating the performance impact of these misses. While store prefetching is demonstrated to be effective, it consumes substantial L2 cache bandwidth, which will be a precious resource in future aggressive chip multi-processors. This paper proposes a new microarchitecture mechanism, the Store Miss Accelerator, to achieve similar gains as store prefetching while conserving L2 cache bandwidth.

Even with store prefetching, the performance impact of off-chip store misses is not fully mitigated and increasing store buffer and store queue sizes is shown to be much less effective in improving MLP than previously assumed. Instead, serializing instructions (e.g. `casa` and `membar` in the SPARC ISA) are the major impediment. Most of these serializing instructions occur in the lock acquire and lock release of critical sections and they are a major cause of the significant performance gap between the processor consistency and weak consistency models. Based on this insight, two techniques, Speculative Lock Elision [13], and transactional memory [14] are evaluated for their efficacy in bridging this performance gap. While these two techniques have been studied in other contexts, their use to improve store performance is novel.

This paper also evaluates Hardware Scouting [15] and Runahead Execution [16,17], recent microarchitecture techniques which have been shown to be highly effective in improving load MLP and instruction MLP [8], for their effectiveness in improving store MLP and for bridging the performance gap between the memory consistency models. Finally, a new microarchitecture optimization, invoking Hardware Scouting when the store buffer is full, is shown to further improve store MLP as well as load and instruction MLP. With this optimization, the performance impact of off-chip store misses is almost completely eliminated.

This paper is organized as follows. Section 2 provides the reader with a background of how stores are executed in a processor and the microarchitecture structures that are required to support their execution. Section 3 describes the epoch MLP model and the extensions added to model store MLP. The model is then used to explain how off-chip store misses interact with other off-chip misses. It is also used to explain how store handling optimizations and the underlying memory consistency model affect these interactions. Section 4 describes the experimental methodology employed and Section 5 presents experimental results. Section 6 describes related work while Section 7 summarizes the contributions of this paper and presents its conclusions.

2 Processor Handling of Stores

This section describes how stores are handled in a processor and provides the background for understanding how stores affect processor performance. Unless indicated otherwise, the system enforces the processor consistency (PC) memory model [11] which requires that stores are globally visible in program order and thus enforces in-order commit of stores into the L2 cache. The L2 cache is shared by all the cores on the Chip Multi-Processor (CMP) and each core has its own write-through no write-allocate L1 data cache.

Two main structures are used to hold stores: store buffer and store queue. A store instruction is allocated a store buffer entry when it is renamed and dispatched. The execution of the store may be split into two distinct operations: the address generation operation calculates and enters the address into the store buffer and the data operation enters the store data into the store buffer. Subsequent loads may safely speculate above stores whose address has been calculated. When both store address and data operations are complete and all earlier instructions have retired, the store is retired. At this point, the store address and data are moved into the store queue. A store is *committed* when the store value is written into the L2 cache and becomes globally visible to all other cores in the system. Thus, a store is held in the store buffer between rename and retirement and in the store queue between retirement and commit.

A straightforward implementation of the PC model serially commits each store when it reaches the head of the store queue. This limits the store bandwidth to, say, one every 10 cycles assuming an L2 cache latency of 10 cycles. In order to sustain a higher bandwidth while still adhering to the PC model, most processors pipeline stores into the L2 cache. In the first phase, the store address is sent to the appropriate L2 cache bank. In the second phase, the L2 cache bank locks the cache line preventing other writes but not reads, and sends an acknowledgment back to the requesting processor. In the final phase, the requesting processor sends the data to the L2 cache. During the first phase, core-to-L2 network resources are reserved so that the latency of the final phase is known to the requesting processor. Thus, multiple stores can be in flight simultaneously and the processor can sustain a bandwidth of up to one store every cycle. On a missing store, the processor receives a negative acknowledgment from the L2 cache bank and the processor cancels all subsequent outstanding stores to maintain store ordering. However, if any of these outstanding stores also miss the L2 cache, the L2 cache may optionally issue prefetches to acquire ownership of the associated lines.

The sizes of the store queue and the store buffer impact processor performance. When the store queue is full, the processor must stop retirement as soon as the next instruction to retire is a store. At that point, the reorder buffer as well as the store buffer can no longer drain, so they begin to fill up. When the reorder buffer is full or when the processor tries to dispatch a store and the store buffer is full, the processor can no longer dispatch any more instructions. Eventually, the processor pipeline is stalled and remains so until the store queue drains. Thus, a missing store at the head of the store queue can stall the processor for several hundred cycles. Note that in the absence of missing stores, i.e. those that require off-chip accesses, the store queue is unlikely to become full since most processors pipeline the commit of stores. Therefore, this study focuses on off-chip store misses.

There are limits to increasing the store buffer and store queue sizes. If an intra-processor data dependence exists between an uncommitted store and a subsequent load, the memory contains a stale value. The processor must detect this dependence and either deliver the value from the store buffer/queue or stall the load until the value is committed. Thus, every load must associatively search the store buffer/queue and ascertain that there are no prior stores with matching addresses. In order to detect store-load dependences, the store buffer/queue are traditionally implemented using a Content Addressable Memory (CAM) structure. The CAM nature of these two structures place a limit on how much they can be enlarged before they impact the processor's clock frequency target. As a result, there has been recent work [18,19] on scaling up the store buffer and store queue by alternative means.

3 Epoch MLP Model

The epoch memory-level parallelism (MLP) model [8] was previously used to study the impact of microarchitecture on the overlap of off-chip load misses (i.e. missing loads) and off-chip instruction misses (i.e. missing instructions). This model assumed that stores are completely overlappable, but the results in this paper indicate that the assumption is valid for the weak consistency (WC) memory model but not for the processor consistency (PC) model. In this paper, the epoch MLP model is extended to model missing stores more accurately. This enhanced epoch MLP model clearly illustrates and accurately quantifies how store-handling mechanisms and optimizations, as well as the processor's memory consistency model, affect the ability of missing stores to overlap with other missing stores and with missing loads and missing instructions.

3.1 Epoch MLP Model

With off-chip latencies of several hundred cycles, on-chip computation (i.e. data computations, address computations, as well as instruction fetches and loads that hit in the on-chip caches) latencies separating overlappable off-chip accesses become increasingly insignificant relative to off-chip access latencies. In such a situation, illustrated in Figure 1, overlappable off-chip accesses appear to issue and complete at the same time, and instruction execution tends to separate itself into recurring periods of on-chip computations and off-chip accesses. Each such period of on-chip computations followed by off-chip access(es) is called an *epoch*. More precisely, an epoch is a time slice of program execution starting from the end of the previous epoch through the first off-chip access and extending to the cycle when this first access completes. Within an epoch, all overlappable off-chip accesses are assumed to issue and complete at the same time. The instruction that is responsible for the first off-chip access of an epoch is called the *epoch trigger*. The set of instructions from the processor's dynamic instruction stream (DIS) that can be executed (committed in the case of stores) in an epoch is called the *epoch set*. The entire DIS is partitioned into epoch sets such that all instructions in epoch set *i* execute in the *i*th epoch.

A *window termination* condition is a condition that prevents the processor from executing any subsequent instructions in the DIS until all earlier off-chip accesses are resolved. Therefore, window termination conditions, together with data dependences, determine epoch sets and which off-chip misses can execute in an epoch.

To illustrate the concepts of epoch, epoch set, epoch trigger and window termination condition, consider the following two example code sequences. In these examples, both the store queue and store buffer are assumed to be two entries deep each, and processor consistency is assumed.

In Example 1, it takes two epochs to execute the six instructions. In the first epoch, the epoch trigger is missing store I1. The PC memory model prevents subsequent stores from committing until I1 commits. The window termination condition is store buffer full (which is preceded by the store queue becoming full), which prevents I5 and I6 instructions from executing in this epoch. The epoch set is therefore {I1}. In the second epoch, I2, I3 and I4 commit and I5 and I6 execute. In Example 2, it also takes two epochs to execute the three instructions. In the first epoch, the epoch trigger is missing store I1. The window termination condition is serializing instruction I2, which requires the processor pipeline to drain before it can execute. The epoch set is therefore {I1}. I2 and I3 execute in the second epoch. Serializing instructions are discussed in more detail in Section 3.3.4. These two examples demonstrate the primary pathways through which store misses lead to processor stalls. In the first example, store misses cause the store queue to back up which leads to either the issue window or the store buffer backing up, eventually leading to a stall in the rename unit. In the second example, a store miss followed by a serialization instruction (such as `casa`, `membar` in SPARC) require that the processor stall until the store miss is committed.

Other than the store buffer full and serializing instruction

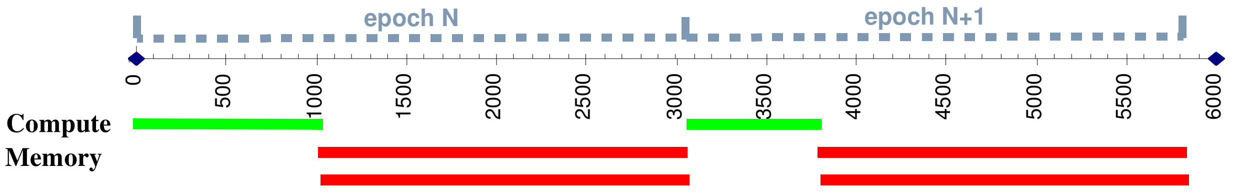

Figure 1. Example where Off-Chip Latency is Much Greater than On-Chip Latencies.

Inst#	Instruction	Comment
I1	missing store	
I2	other store	<-StQ Full
I3	other store	
I4	other store	<-StBuf Full
I5	other store	
I6	missing load	

Epoch Sets =
{ {**I1**}, {I2, I3, I4, I5, **I6**} }
MLP = (1+1)/2 = 1

Example 1

Inst#	Instruction	Comment
I1	missing store	
I2	serializing instruction	
I3	missing load	

Epoch Sets =
{ {**I1**}, {I2, **I3**} }
MLP = (1+1)/2 = 1

Example 2

Inst#	Instruction	Comment
I1	missing load	
I2	missing store	
I3	missing instruction	
I4	missing store	

Epoch Sets =
{ {**I1**, ***I3***}, {**I2**, I3}, {**I4**} }
MLP = (2+1+1)/3 = 1.33

Example 3

window termination conditions, the other key window termination conditions are: instruction fetches that miss the L2 cache, unresolvable branch mispredictions, and issue window size and reorder buffer size limitations (for out-of-order issue processors). Moreover, apart from missing stores, missing loads and missing instructions can also be epoch triggers. To illustrate some of these window termination conditions and epoch trigger types, consider Example 3, where the epoch trigger in the first epoch is missing load I1 and the window termination condition is the instruction fetch miss at I3. Because I1 remains at the head of the reorder buffer (ROB) until its miss completes (i.e. at the end of the first epoch), I2 cannot retire and commit until the next epoch. With processor consistency, I4 cannot commit until I2 commits. So in the absence of store prefetching, I4 commits in the third epoch.

Unlike missing loads and missing instructions, whose latencies are only marginally overlapped with on-chip computation [8], the latencies of some missing stores can be fully overlapped with on-chip computation. These are the missing stores where the processor does not incur any stall due to an off-chip access while they are resident in the store queue. Specifically, the processor does not stall because the store queue did not become full and the processor did not encounter a missing load, a missing instruction or a serializing instruction. In the epoch model, these missing stores never result in additional epochs. Table 2 shows the fraction of missing stores that are fully overlapped with computation for the processor configuration described in Section 4.3 (assuming a memory latency of 500 cycles). These results show that most missing stores cannot be overlapped with computation and the importance of mitigating their performance impact by overlapping them with other missing stores or with missing loads and missing instructions.

Table 2: Fraction of missing stores fully overlapped with computation.

Database	TPC-W	SPECjbb	SPECweb
0.09	0.12	0.06	0.22

3.2 MLP Definitions

In [8], MLP was defined as *the average number of useful long-latency off-chip accesses outstanding when there is at least one such access outstanding*. In the presence of missing stores, this definition can be retained by expanding the definition of off-chip accesses to include missing stores in addition to missing loads and missing instructions. In the epoch model, MLP is simply the ratio of the total number of off-chip accesses to the number of epochs. It is also convenient to define another metric, store MLP, as *the average number of useful long-latency off-chip store accesses outstanding when there is at least one such access outstanding*. Store MLP is a measure of the ability of missing stores to overlap with other missing stores.

3.3 Store Optimizations

We now use the epoch model to describe how store optimizations impact the ability of missing stores to overlap with other misses. Some of these optimizations such as store coalescing and store prefetching have been previously proposed. Other optimizations, such as the Store Miss Accelerator and the optimization for Hardware Scouting, are proposed here for the first time. The memory consistency model optimizations, transactional memory [14] and Speculative Lock Elision (SLE) [13] have been studied in other contexts but their evaluation in the context of improving store performance is novel.

3.3.1 Store Coalescing

In order to conserve store queue entries, retiring stores can be combined with prior stores as they enter the store queue. For instance, with a coalescing granularity of 8 bytes, two stores can potentially be coalesced if they both write the same 8 byte aligned memory location. The PC model only allows consecutive stores to be coalesced, while the WC model generally allows a retiring store to be coalesced with any eligible store in the store queue. Store coalescing improves store performance by increasing the effective capacity of the store queue and reducing the fraction of time that it is full.

3.3.2 Store Prefetching

To increase store MLP, the processor can perform hardware prefetching of stores. We identify two hardware store prefetching schemes: *prefetch at retire* [20] and *prefetch at execute* [21]. For prefetch at retire, when a store retires and the store queue is not empty, a "prefetch for write" request is sent to the L2 cache for that store. As a result, the latency of all the missing stores in the store queue can be overlapped and they can commit in the same epoch. For prefetch at execute, the prefetch for write request is sent as soon as the store generates the store address. Accordingly, the latency of all the missing stores in the store queue and the store buffer can be overlapped with each other and all stores can commit in the same epoch. Therefore, prefetch at execute can potentially achieve higher store MLP than prefetch at retire.

To illustrate the effects of store prefetching, consider Example 4. In the absence of store prefetching, I1, I2 and I3 commit in separate epochs, thereby requiring a total of three epochs for the processor pipeline to drain. With prefetch at retire, I1 and I2 commit in the same epoch while I3 commits in the second epoch. With prefetch at execute, I1, I2 and I3 can all commit in one epoch.

A drawback of store prefetching is that it consumes additional L2 cache bandwidth because two write requests may potentially be issued for every store. Compared to prefetch at execute, the L2 cache bandwidth overhead is potentially smaller for prefetch at retire because prefetches do not need to be issued for the coalesced stores. Prefetch at retire also does not issue stores that are on the wrong control flow path.

3.3.3 Store Miss Accelerator

In aggressive CMP designs [22], a key bottleneck is the interconnection network between the cores and the shared L2 (or last on-chip level) cache. Store prefetching is effective but requires a significant amount of core-to-L2 bandwidth. In this section, we propose a novel caching scheme directed at accelerating store misses without utilizing valuable core-to-L2 bandwidth.

Consider a single-chip system where a number of cores share a single L2 cache. In this system, the single L2 cache implicitly owns the entire memory in exclusive mode, except possibly for pages that are shared with I/O devices. Thus, on an off-chip store miss, the L2 cache may buffer the store miss data, enabling the missing store and subsequent stores to leave the store queue before the rest of the line is available from memory. The L2 cache eventually merges the store miss data with the rest of the line from memory. Store ordering as required by processor consistency is preserved because the missing store data is globally visible to the other cores even before the data is merged with the rest of the cache line.

In a multi-chip system, missing stores can be made globally visible only after the line is known to be invalid in other chips. As a result, missing stores incur the invalidation penalty before they can leave the store queue. However, most missing stores access private data that are repeatedly brought into the L2 cache, modified and then evicted (due to capacity constraints). The underlying problem is that when the modified line is evicted, not only does the L2 lose the data itself but also the ownership of that line. A key observation is to decouple maintaining ownership from maintaining the data value of the line. Maintaining ownership requires only a few of bits of state whereas maintaining the line value requires over 500 bits of state, assuming a cache line size of 64 bytes. If the L2 subsystem has ownership of the requested lines, missing stores do not need to incur the invalidation penalty.

For the purpose of this discussion, the MESI protocol is assumed, although the scheme can be easily extended to the MOESI protocol and to either broadcast or directory-based protocols. The Store Miss ACcelerator (SMAC) holds additional lines (beyond those already in the L2 cache) in the exclusive (E) state. When a modified (M) line is evicted from the L2 cache, the line is written to memory, and the downgraded E state is held in the SMAC. On a snoop (either a request-to-own or shared) from another chip that hits in the SMAC, the line is invalidated. On a store miss in the L2 cache and a hit of this line in the SMAC, the store miss is allowed to proceed without incurring any invalidation penalty as in a single-chip system, since this chip already has the line in E state. Thus, the store miss incurs the invalidation penalty only when it misses the SMAC. In Example 4, assume that I2 and I3 hit in the SMAC even though they miss in the L2 cache. In this case, all three stores can proceed in the same epoch.

The SMAC can be designed to cover a much larger memory address space than the L2 cache. In order to amortize the cost of maintaining the tag and take advantage of the spatial locality of store misses, the SMAC is organized as a heavily sub-blocked set-associative cache with E state indicated by one bit per L2 cache line. For example, a SMAC can cover 16 MB of the address space with 8K tags and 2048-byte lines that are 32-way sub-blocked. Assuming a 40-bit physical address, the tag size is 32-bits and the total size of the SMAC is 8K * (32 + 32)/8 or 64 KB. The SMAC is located in the L2 subsystem and can have relatively slow access time similar to the L2 tag access latency.

3.3.4 Memory Consistency Model Optimizations

The underlying memory consistency model supported by the processor strongly impacts store performance. In this paper, we consider two classes of memory consistency models: processor consistency (PC) [11] and weak consistency (WC) [12]. For a concrete example of the former, we use the Sun SPARC Total Store Order (TSO) [23] model. For the latter, we use the IBM PowerPC architecture [24] weakly consistent (WC) model that is employed in the IBM Power4 and Power5 servers. Due to space constraints, we confine our discussion of these models to their impact on stores.

In contrast to PC, WC permits stores to commit out-of-order and a missing store at the head of the store queue does not prevent subsequent stores that hit the L2 cache from releasing their store queue entries. Consequently, the store queue is unlikely to become full unless there is a strong clustering of missing stores. In the previously shown Example 1, if the processor implements WC, stores I2, I3, I4 and I5 can commit even while the missing store I1 is waiting to commit. As a result, the store queue and the store buffer do not become full and the missing load I6 can issue in the first epoch, reducing the number of epochs from two to one. Furthermore, WC improves the effectiveness of store coalescing because a retiring store may be coalesced with any eligible store in the store queue.

Critical sections require memory operations within them to be performed after the lock acquisition and before the lock release. Critical sections impact store performance because they are usually implemented using serializing instructions, which often require the store queue and the store buffer to drain before they can execute. Consider two alternate implementations of critical sections using the PC and WC models. In the PC model (Example 5), the `casa` performs a load and store atomically. Since loads are ordered and stores are ordered, no memory operation inside the critical section will be performed before the lock acquisition. At the end of the

Inst#	Instruction	Comment
I1	missing store	
I2	missing store	<-StQ Full
I3	missing store	
I4	serializing instr.	

No Store prefetching
{{**I1**}, {**I2**}, {**I3**}}
Prefetch at Retire
{{**I1**, **I2**}, {**I3**}}
Prefetch at Execute
{{**I1**, **I2**, **I3**} }
Example 4

Inst#	Instruction	Comment
I1	missing store	
I2	casa	lock acquire
I3	missing load	
I4	missing store	
I5	...	
I6	store	lock release
I7	missing load	

Epoch Sets =
{{**I1**}, {I2, **I3**, **I4**, **I7**}, {I6}}
Example 5

Inst#	Instruction	Comment
I1	missing store	
I2	lwarx/stwcx	lock acq.
I3	isync	
I4	missing load	
I5	missing store	
I6	lwsync	
I7	store	lock rel.
I8	missing load	

Epoch Sets =
{{**I1**, I2, I3, **I4**, **I5**, **I8**},
{I6, I7} }
Example 6

critical section, a store releases the lock. Again, since stores are ordered and since stores cannot speculate past loads, all memory operations in the critical section are performed before lock release. In the WC model (Example 6), the lock acquire is usually implemented using the load-locked (`lwarx`) and store-conditional (`stwcx`) pair [24] and the `isync` [24] barrier instruction ensures that the lock acquisition is complete before critical section instructions ultimately execute. The `lwsync` barrier ensures that loads/stores above the barrier are performed before stores after the barrier and therefore that load/stores within the critical section are performed before the lock release.

The PC model enforces additional orderings on memory operations outside the critical section that are not essential. The `casa` forces earlier stores to be performed before entering the critical section. Thus, all missing stores in the store buffer/queue have to drain before the `casa` can be executed. In contrast, the `isync` instruction in the WC model does not enforce waiting for the store queue and store buffer to drain. The lock release constraints are similar in both the PC and WC models. Both unnecessarily force stores after the critical section (other than the lock release) to be performed after the memory operations in the critical section, but this constraint has negligible impact on performance. Both permit loads after the critical section to speculate and be performed before memory operations in the critical section.

As can be seen from the above description, the key difference between the typical critical section implementations of the two consistency models with regard to store performance is that the WC model does not require the store buffer and store queue to drain prior to executing the critical section. As the results in this paper demonstrate, this difference leads to a significant performance gap between the two models. To bridge this gap, we propose exploiting Speculative Lock Elision (SLE) [13]. SLE was originally proposed as a microarchitecture technique to improve the performance of applications which experience heavy lock contention but not data contention. In this technique, the lock is not acquired and the critical section is entered speculatively. At the exit of the critical section, if no other thread in the system tried to load or store the same data that was loaded or stored in the critical section, the speculation succeeded and the operations in the critical section can be committed. SLE effectively converted the lock acquire into a regular load and the lock release into a NOP. Other than improving store performance, SLE also improves load and instruction MLP since the lock acquire is no longer serializing. While SLE improves the performance of weak consistency, it is even more beneficial for processor consistency. A related technique, transactional memory [14] achieves similar benefits as SLE but requires software as well as hardware support.

Finally, to mitigate the performance impact of the `casa` instruction in the TSO model and the `isync` instruction in the PowerPC model, the processor can prefetch the loads and stores beyond these instructions. The number of loads and stores that can be prefetched is limited by the size of the reorder buffer since the `casa` and `isync` instructions usually hold up instruction retirement.

3.3.5 Hardware Scouting Optimization

Execution-based prefetching [2] such as Hardware Scouting (HWS) [15] and Runahead Execution [16,17] have been shown to be effective techniques for enhancing load and instruction MLP [8]. In a conventional out-of-order processor, a missing load at the head of the reorder buffer (ROB) blocks instruction retirement, so the number of subsequent missing loads and missing instructions that can enter the ROB/instruction window and be overlapped is restricted by the sizes of these structures. In HWS, when the missing load epoch trigger becomes the oldest entry in the ROB, the processor checkpoints the register files and enters scout mode. In this mode, all missing loads are converted to prefetches and do not stall the pipeline. Any subsequent instructions that are dependent on these missing loads are simply skipped. Also, stores do not update architected state. When the data of the missing load epoch trigger returns, the processor flushes its pipeline, restores the register checkpoint and re-enters normal execution mode. Since HWS is purely speculative, it does not have to obey the serialization constraints of serializing instructions. Other than enhancing load and instruction MLP, HWS can also increase store MLP if prefetches are issued for the stores encountered in scout mode. While previous evaluations of this technique only entered scout mode on a missing load, we propose also invoking HWS when the store queue is full and rename/dispatch is stalled. The results in Section 5.4 show that this optimization of HWS is highly effective in improving store MLP.

3.4 Epochs and Overall Performance

The number of epochs is related to overall processor performance in the following manner. The total execution time of an application, $\text{Cycles}_{\text{total}}$, is given by the following equation:

$$Cycles_{total} = Cycles_{on\text{-}chip}(1 - Overlap) + Cycles_{off\text{-}chip}$$

where $\text{Cycles}_{\text{on-chip}}$ is the number of cycles spent in on-chip computation, $\text{Cycles}_{\text{off-chip}}$ is the number of cycles spent in off-chip accesses, and Overlap is the fraction of on-chip cycles that were overlapped with off-chip cycles. $\text{Cycles}_{\text{on-chip}}$ can be measured on a cycle simulator by assuming that the furthest on-chip cache is perfect. In the epoch model, $\text{Cycles}_{\text{off-chip}}$ is the product of the number of epoch and the off-chip miss penalty. Therefore:

$$Cycles_{total} = Cycles_{on\text{-}chip}(1 - Overlap) + (Number\ of\ Epochs \times MissPenalty)$$

Alternatively, the above equation can be expressed in terms of overall clocks per instructions (CPI):

$$CPI_{overall} = CPI_{on\text{-}chip}(1 - Overlap) + (Epochs\ Per\ Instruction \times MissPenalty)$$

The second term of this equation is the off-chip CPI. In the comparison of the various store handling mechanisms, it is convenient and insightful to use the Epochs Per Instruction (EPI) metric because this metric has a linear relationship with off-chip CPI. In addition, since Overlap varies very little over the different store handling mechanisms, it can be approximated as a constant. In that case, EPI also has a linear relationship with overall CPI.

For the processor configuration described in Section 4.3, and assuming an L1 cache latency of 4 cycles and L2 cache latency of 15 cycles, $CPI_{on-chip}$ is as shown in Table 3.

Table 3: $CPI_{on-chip}$ for default processor configuration.

Database	TPC-W	SPECjbb	SPECweb
1.11	1.12	0.95	1.38

4 Experimental Methodology

4.1 Performance Metrics and MLPsim

The primary performance metric used to evaluate and compare various store optimization schemes is Epochs Per Instruction (EPI). As described earlier, off-chip CPI is the product of EPI and the miss penalty. Thus, EPI directly relates to the familiar CPI performance metric but is independent of on-chip latencies and miss penalty. The EPI metric is valuable in exploring high-level design choices, analogous to the manner in which cache miss rates are used to characterize cache designs in a high-level exploration of the design space. For in-depth analysis of missing store behavior, the MLP metrics described in Section 3.2 are also useful. In contrast to CPI metrics, which are susceptible to the choice of detailed parameter settings and have proven in the past to be difficult to reproduce, the EPI and MLP metrics are easier to reproduce and validate.

In a cycle-accurate simulator, EPI is tracked by counting epoch triggers. Recall that the epoch trigger is the first off-chip cache miss in an epoch. Accordingly, the number of times the number of outstanding off-chip misses transitions from 0 to 1 is counted. MLP is measured by averaging the number of misses outstanding over all cycles where at least one miss is outstanding. These metrics may be adjusted for store misses that are completely overlapped with computation.

While our in-house cycle-accurate simulator is fully capable of measuring EPI and MLP, we choose to present the simulation results from MLPsim in this study. MLPsim is a MLP simulator that implements the epoch MLP model. It reads in an instruction trace and a set of microarchitecture parameters as inputs, and outputs MLP and epoch statistics. It partitions the instruction trace into epoch sets by tracking register and memory dependences between instructions, by modeling the sizes of key hardware structures such as the fetch buffer, issue window, reorder buffer, store buffer and store queue, and by applying the window termination conditions associated with the microarchitecture parameters and memory consistency model specified. For the purposes of this study, MLPsim is preferable to a cycle-accurate simulator because it abstracts away many microarchitecture features that do not have appreciable impact on store performance, thereby providing a clearer insight into the effects and interactions of those features that do. Even more importantly, MLPsim provides deep insight into the extent of overlap between missing stores and other off-chip misses, which is the key to understanding store performance.

4.2 Benchmarks

Four commercial workloads are used in this study: a database workload, TPC-W, SPECjbb2000, and SPECweb99. The database workload is a full-scale online transaction processing set-up that involves hundreds of disks. TPC-W is a transactional web benchmark that simulates the activities of a business oriented transactional web server. SPECjbb2000 is a server-side Java benchmark that emphasizes business logic and object manipulation, the middle tier of a 3-tier system. SPECweb99 is another benchmark for measuring web server performance, but unlike TPC-W, it does not use transactions. The Sun Java System Application Server was used in running TPC-W and the Sun Java System Web Server was used in running SPECweb99. For SPECjbb2000, the Sun Java 2 SDK 1.4.2 was used.

The highly optimized SPARC binaries used to generate traces were compiled for the TSO memory consistency model (which is a PC model). The traces were generated by a full-system simulator simulating a multiprocessor and they contain both kernel and user instructions. The traces were collected when the workloads were warmed and running in steady state.

In all the experiments, for each core, the first 50M instructions in the trace are used to warm the caches and the next 100M instructions (sufficient due to the absence of phase behavior) are used to collect statistics. Since the SMAC covers a larger memory space, the simulation runs involving the SMAC used 1B instructions for warming. In the simulations, the cross-chip coherence traffic is accurately modeled.

To simulate weak consistency using processor consistency traces, a lock detection tool was developed to identify all the lock acquisition and lock release instruction sequences in the traces. These instruction sequences were then replaced with the appropriate instruction sequences and barriers.

4.3 Default Processor Configuration

The simulations assume a 2-way multiprocessor. Each processor comprises of two single-threaded cores sharing an L2 cache. Unless stated otherwise, they assume the following default core configuration.

Private L1 (I + D)	32KB 4-way set associative, LRU, 64B lines
Shared L2	2MB 4-way set-associative, LRU, 64B lines
Shared TLB	2K entries
Branch prediction	64K entry gshare, 16K entry BTB, 16 entry RAS
Front-end	32 entry fetch buffer
IW/ROB	32 entry instruction window, 64 entry ROB,
Issue policy	Out-of-order, loads issue OOO wrt other loads and stores and speculate past earlier unresolved stores, branches issue OOO wrt other branches
Store handling	16 entry store buffer, 32 entry store queue, store prefetch at retire, 8 byte store coalescing
Load handling	64 entry load buffer
Memory model	Processor consistency

5 Experimental Results

In Sections 5.1 and 5.2, the effects of store queue size, store buffer size, store prefetching, store coalescing and the Store Miss Accelerator are presented for the processor consistency model. In Section 5.3, the performance of processor consistency is contrasted with weak consistency and techniques for bridging their performance gap are evaluated. In Section 5.4, the effectiveness of Hardware Scouting and its optimizations is evaluated.

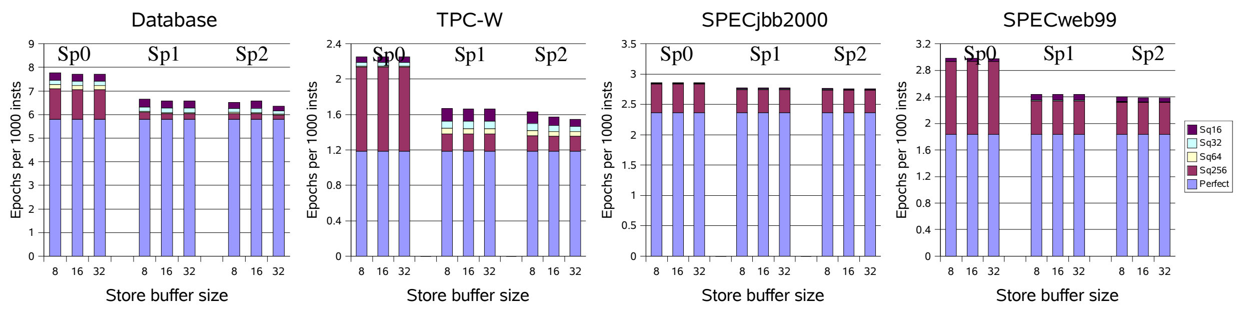

Figure 2: Effectiveness of store prefetching, store buffer size and store queue size.

5.1 Store Queue Size, Store Buffer Size, Store Coalescing and Store Prefetching

Figure 2 shows the effects of store buffer size, store queue size and store prefetching for the PC model. Store coalescing at a granularity of 8 bytes is assumed. For each graph, there are three sets of bars (Sp0, Sp1 and Sp2), corresponding to no store prefetch (Sp0), prefetch at retire (Sp1) and prefetch at execute (Sp2) respectively. Within each set, there are three bars, corresponding to store buffer sizes of 8, 16 and 32 respectively. For each bar, the height of the bottom segment is the number of epochs per thousand instructions if stores never stalled the processor, while the combined height of the bottom segment and the next segment is the number of epochs per thousand instructions for a store queue size of 256. The combined height of all the segments is the number of epochs per thousand instructions for a store queue size of 16. The intermediate heights correspond to store queue sizes of 64 and 32. As mentioned in Section 3.4, the epochs per instruction metric can be translated into off-chip CPI simply by multiplying it with the off-chip latency. For example, if the off-chip latency is 500 cycles, 5 epochs per 1000 instructions translate to an off-chip CPI of 2.5. Thus, a lower bar height corresponds to better performance.

The key results are as follows. First, for all workloads except SPECjbb2000, store prefetching (either prefetch at retire or prefetch at execute) is highly effective at mitigating the performance impact of missing stores. Second, for SPECjbb2000 and SPECweb99, even with store prefetching enabled, missing stores still severely impact performance. For these workloads, enlarging the store queue or store buffer has no effect because the key performance limiter is actually serializing instructions (see explanation of Figure 3). For the database workload and TPC-W, store queue size does impact performance but serializing instructions are also a performance limiter. Third, for all four workloads, store MLP is not sensitive to the store buffer size. For the chosen reorder buffer size of 64 entries, eight entries are sufficient for the store buffer.

An examination of Figure 2 reveals that in the absence of store prefetching (configuration Sp0, Sb16, Sq32), the contribution of missing stores to off-chip CPI is 22% for the database workload, 46% for TPC-W, 17% for SPECjbb2000 and 37% for SPECweb99. With store prefetching (default configuration Sp1, Sb16, Sq32), the contribution decreases to 8% for the database workload, 22% for TPC-W, 14% for SPECjbb2000 and 22% for SPECweb99. It is evident that missing stores contribute to a significant percentage of off-chip CPI and must be handled carefully to mitigate their performance impact. It is important to note that for these commercial workloads, as memory latencies increase, off-chip CPI begins to dominate overall CPI [7]. Therefore, optimizing store performance is important for optimizing overall performance.

Due to space constraints, the results of the store coalescing experiments are not shown but are briefly described here. Store coalescing is moderately effective for the database workload and TPC-W when the store queue size is 32 entries or smaller. For the database workload, coalescing at 64 bytes granularity enables a store queue size of 32 to perform as well as a store queue size of 64 without store coalescing. However, store coalescing has no effect for SPECjbb2000 and SPECweb99 since store queue size is not their performance limiter. Overall, store coalescing has a much smaller performance effect than store prefetching.

Figure 3A shows the relative frequency of various window termination conditions for the default processor configuration. The key observation is that for TPC-W, SPECjbb and SPECweb, *store serialize* is the dominant window termination condition. For these workloads, the missing stores preceding

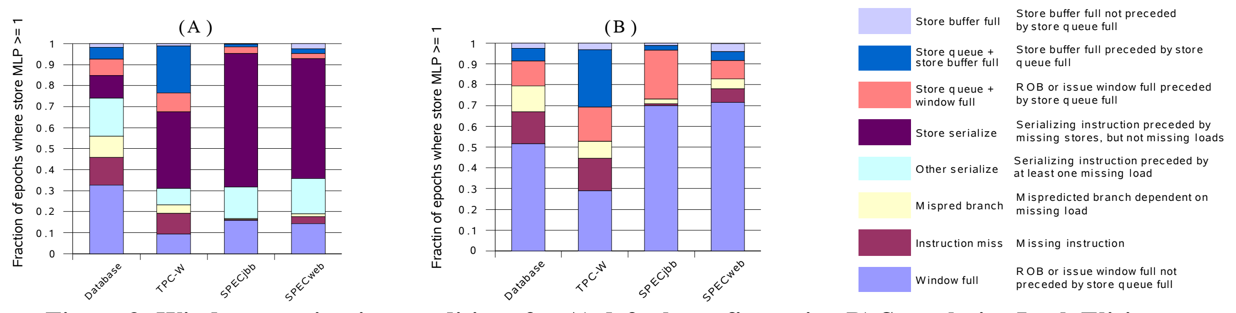

Figure 3: Window termination conditions for A) default configuration B) Speculative Lock Elision.

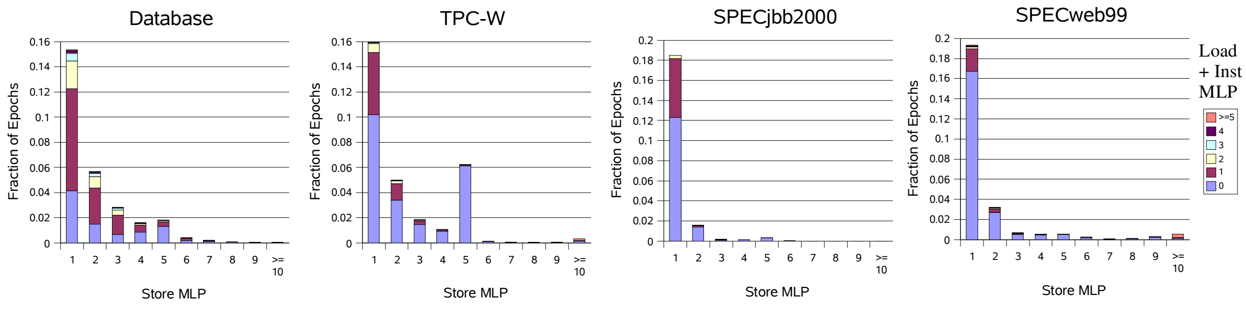

Figure 4: MLP distributions for default configuration.

the serializing instruction cannot be overlapped with other off-chip misses and therefore their latencies are exposed. For SPECjbb2000 and SPECweb99, serializing instructions rather than store queue size is the real limiter to store performance.

Figure 4 shows the extent to which missing stores can be overlapped with other missing stores and with missing loads and missing instructions for the default processor configuration. In each graph, the total height of each bar represents the fraction of total epochs with that amount of store MLP. The sum of the heights of all the bars does not equal one because the bar for zero store MLP is not shown. Within each bar, the different segments represent varying amounts of combined load and instruction MLP. For example, the bottom segment of the left-most bar corresponds to the fraction of total epochs where there is one missing store and no missing loads or missing instructions. These missing stores are most expensive as they are not overlapped with any other off-chip misses.

For the database workload, there are relatively few expensive missing stores. Most missing stores can either be overlapped with other missing stores or with at least one missing load or missing instruction. However, for TPC-W, SPECjbb2000 and SPECweb99, expensive missing stores are prevalent. For SPECjbb2000 and SPECweb99, the majority of missing stores cannot be overlapped with any other missing store or with any missing loads or missing instructions. These missing stores precede serializing instructions.

5.2 Store Miss Accelerator

Figure 5 shows the effectiveness of the Store Miss Accelerator (SMAC) in improving store performance. In this figure, each graph comprises of three bars, corresponding to no store prefetch (Sp0), prefetch at retire (Sp1) and prefetch at execute (Sp2) respectively. For each bar, the height of the bottom segment is the number of epochs per thousand instructions if stores never stalled the processor, while the combined height of the bottom segment and the next segment is the number of epochs per thousand instructions for a SMAC with 128K entries. The combined height of all the segments is the number of epochs per thousand instructions in the absence of a SMAC (i.e. the default configuration). The intermediate heights correspond to SMAC sizes of 64K entries, 32K entries, 16K entries and 8K entries. Note that in our implementation, each SMAC entry occupies 64 bits (8 bytes).

The results show that the SMAC is very effective in improving store performance. In the absence of store prefetching, for the database workload, a 64K entry SMAC is able to achieve similar effect as store prefetch at execute. For SPECjbb2000 and SPECweb99, 32K entries and 16K entries are sufficient respectively. In the presence of store prefetching, the SMAC is able to mitigate the performance impact of serializing instructions by removing a large fraction of the missing stores that immediately precede a critical section.

Figure 6 shows how often invalidates from other processors impact the effectiveness of the SMAC. The left graph shows the number of such invalidates per 1000 instructions as the number of SMAC entries and the number of nodes in the multiprocessor are varied. The right graph shows the percentage of missing stores that find a matching entry in the SMAC but the entry was invalidated due to a coherence event caused by another node. The results show that the SMAC performs well even as the number of nodes in the multiprocessor is scaled up.

5.3 Memory Consistency Model

In the results presented so far, the processor consistency model has been assumed. Figure 7 compares the performance of this memory consistency model with weak consistency. In this figure, each graph comprises of three sets of bars, corresponding to no store prefetch (Sp0), prefetch at retire (Sp1) and prefetch at execute (Sp2) respectively. Within each set,

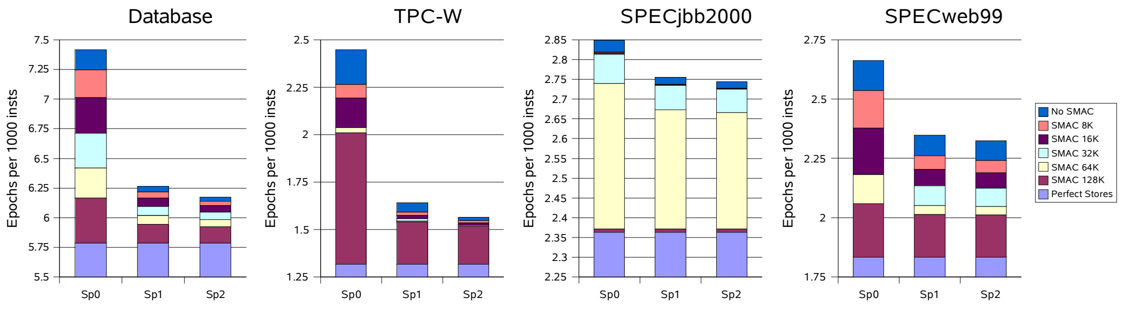

Figure 5: Performance effects of Store Miss Accelerator.

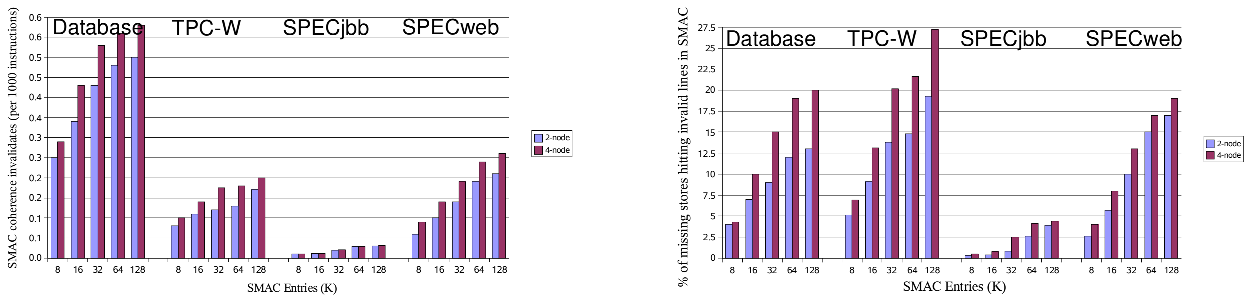

Figure 6: Impact of coherence events on SMAC effectiveness.

each bar indicates a simulated configuration. Six configurations were simulated, the first three for the processor consistency model and the last three for the weak consistency model.

Processor consistency:

PC1 - default processor configuration

PC2 - prefetch past serializing instruction

PC3 - SLE plus prefetch past serializing instruction

Weak consistency:

WC1 - lock acquire implemented using `isync` instruction, lock release implemented using `lwsync` instruction

WC2 - WC1 plus prefetch past serializing instructions

WC3 - SLE plus prefetch past serializing instructions

Each bar in a graph comprises two segments. The height of the bottom segment is the number of epochs per thousand instructions if stores never stalled the processor, while the combined height of the two segments is the number of epochs per thousand instructions if stores are accounted for.

Comparing the default processor consistency configuration PC1 to WC1, there is a large difference in store performance between the two memory consistency models. To bridge this performance gap, the techniques described in Section 3.3.4, prefetching past serializing instructions and Speculative Lock Elision (SLE), were evaluated. In our SLE experiments, we assumed that all lock elisions are successful.

The key results are as follows. First, SLE is effective in reducing the performance gap between the two memory consistency models for TPC-W, SPECjbb and SPECweb. For processor consistency, it is also very effective in mitigating the performance impact of missing stores. For some workloads, this technique also improves load and instruction MLP. Second, prefetch past serializing instruction improves performance moderately for the database workload and SPECjbb2000. Third, even with SLE and prefetch past serializing instructions, store prefetching is still important.

To analyze the benefits of SLE for processor consistency, Figure 3B shows the window termination conditions for configuration PC3. Compared to Figure 3A, *store serialize* is no longer the dominant window termination condition. In fact, it is negligible for SPECjbb and SPECweb. The *other serialize* window termination condition no longer occurs because this configuration prefetches past serializing instructions. For TPC-C, SPECjbb and SPECweb, *window full* is now the dominant window termination condition. For TPC-W, it is *store queue + store buffer full*. In addition, an analysis of the store MLP distributions (not shown due to space constraints) shows that expensive missing stores are no longer prevalent, even for TPC-W, SPECjbb2000 and SPECweb99. Most stores now can either be overlapped with other missing stores or with at least one missing load or missing instruction.

5.4 Effect on Hardware Scouting

Figure 8 shows the effectiveness of Hardware Scouting (HWS) and its optimizations. Each graph comprises of two

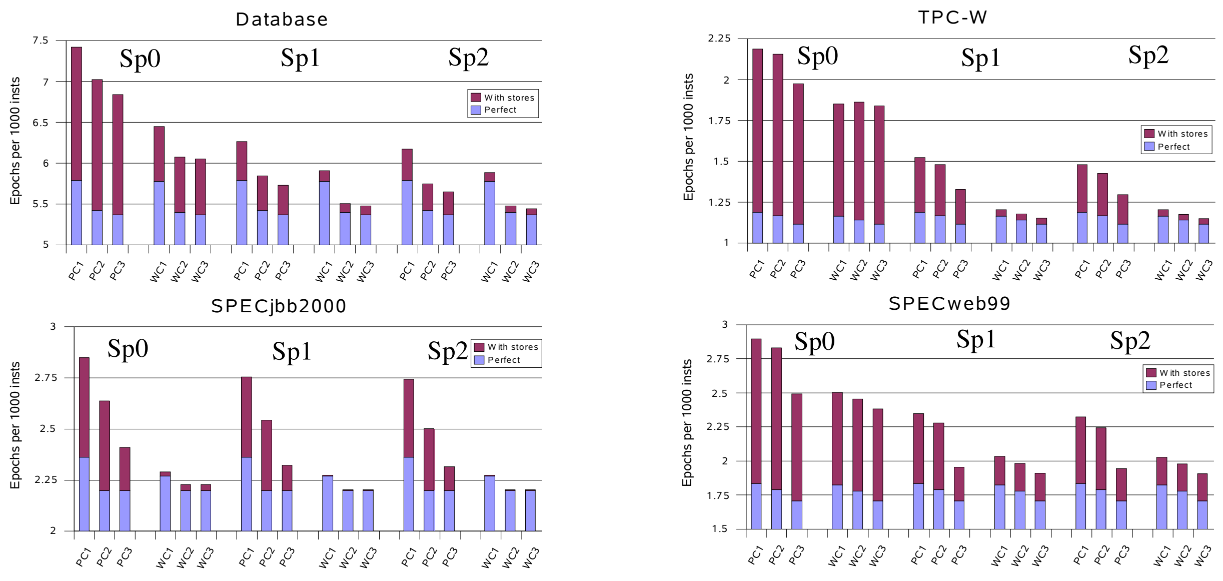

Figure 7: Effectiveness of memory consistency model optimizations.

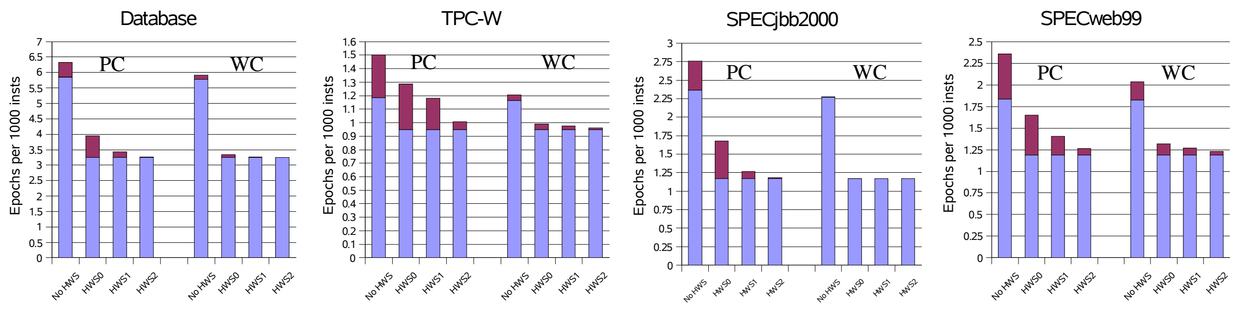

Figure 8: Effectiveness of Hardware Scout optimizations.

sets of bars, corresponding to processor consistency and weak consistency respectively. Within each set, each bar indicates a different configuration.

No HWS - default processor configuration

HWS0 - invoke HWS on missing load, prefetch only for missing loads and missing instructions while in HWS

HWS1 - invoke HWS on missing load, prefetch for missing stores, missing loads and missing instructions while in HWS

HWS2 - invoke HWS on missing load as well as on store queue full, prefetch for missing stores as well as for missing loads and instructions while in HWS

Each bar in a graph comprises two segments. The height of the bottom segment is the number of epochs per thousand instructions if stores never stalled the processor, while the combined height of the two segments is the number of epochs per thousand instructions if stores are accounted for. The results demonstrate that our proposed HWS optimization, HWS2, is effective in almost fully mitigating the impact of missing stores. It also almost completely bridges the performance gap between the two memory consistency models. In general, HWS is also very effective in improving load and instruction MLP.

6 Related Work

While there have been numerous studies [3,4,5,6,7] on the performance of modern processors on commercial applications, none of them examined the performance of stores or the impact of store handling optimizations in detail. Prior research on store handling optimizations such as store prefetching [20, 21], store coalescing [25], the sizing of the store buffer/store queue [26] and cache write policies [27,28] were not performed using commercial workloads.

The performance of stores is strongly impacted by the underlying memory consistency model supported by the processor. There has been substantial research [21,29,30,31] in closing the performance gap between the stronger and weaker models. Two of the early optimizations proposed, hardware prefetch [21] and speculative loads [21], have been adopted in commercial processors such as the Intel Pentium Pro, HP PA-RISC 8000 and the MIPS R10000, and are incorporated in our baseline configuration. To date, most if not all the studies of memory consistency models have been performed using scientific workloads.

In parallel with our work, [32] and [33] propose avoiding unnecessary coherence events by enabling each processor to retain coarse-gain coherence information for large memory regions. Processors accessing data within these regions can potentially avoid long duration snoop operations to ascertain ownership of the requested line. In contrast to these proposals that only remove the snoop overhead of missing stores, the SMAC hides the entire miss latency. Additionally, the SMAC retains coherence information on a much finer grain, which can be useful when regions of memory are shared, but the specific sub-regions are not.

Our previous study [8] used the epoch MLP model to study the impact of microarchitecture on load and instruction MLP. This original epoch model assumed that all stores are fully overlappable. This paper shows that this assumption is reasonable for the WC model but stores have to carefully modeled in the PC model.

Finally, the notion of epochs in our model is similar to the notion of eras in the model presented in [34] but that study was not focused on store performance.

7 Conclusions

In this paper, the impact of stores on processor performance is evaluated for four modern commercial applications. It is shown that there are significant numbers of missing stores in these commercial applications. As the performance impact of stores is largely determined by the ability of missing stores to be overlapped with other missing stores and with missing loads and missing instructions, the epoch MLP model is used to explain how these overlaps are affected by various store handling optimizations and by the memory consistency model implemented by the processor. The extent of these overlaps and their resultant impact on off-chip CPI are then quantified using MLPsim, a simulator that implements the epoch MLP model.

The experimental results show that there is limited overlap between missing stores and computation, and without store handling optimizations such as store prefetching, missing stores have a drastic and negative impact on processor performance. The sizes of the store buffer and store queue are found to be much less critical for store MLP than previously assumed. Instead, serializing instructions are a major impediment to store MLP, especially for SPECjbb2000 and SPECweb99. Most of these serializing instructions occur in the lock acquire and lock release of critical sections and they are a major cause of the performance gap between processor consistency and weak consistency. Unlike a previous study, the experimental results show that this performance gap is substantial for the commercial applications studied. To bridge this performance gap, the use of Speculative Lock Elision is proposed. The experimental results show that it is effective in narrowing the performance gap. In addition, a new optimization, invoking Hardware Scouting when the store buffer is full, is shown to be able to almost completely mitigate the performance impact of missing stores and to almost fully bridge the performance gap between the two memory consistency models. Lastly, a new microarchitecture mechanism, the Store Miss Accelerator, is proposed to improve store MLP while conserving L2 cache bandwidth. The experimental results show that it successfully attains these goals.

Acknowledgments

The authors would like to thank Craig Anderson, Brian Fahs, Peter Lawrence, Harit Modi, Sreenivas Vadlapatla and Jhy-Chun Wang for their infrastructure contributions.

References

[1] L. Spracklen, Y. Chou, and S. G. Abraham, "Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications", Intl. Symp. on High-Performance Computer Architecture, pp. 225-236, 2005.

[2] S. Iacobovici, et. al., "Effective Stream-based and Execution-based Data Prefetching", Intl. Conf. on Supercomputing, pp. 1-11, 2004.

[3] P. Ranganathan, et. al., "Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors", Intl. Conf. on Architectural support for programming languages and operating systems, pp. 307-318, 1998.

[4] L. Barroso, K. Gharachorloo and E. Bugnion, "Memory System Characterization of Commercial Workloads", Intl. Symp. on Computer Architecture", pp. 3-14, 1998.

[5] A. Maynard, C. Donnelly, B. Olszewski, "Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads", Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 145-156, 1994.

[6] J. Lo et. al., "An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors", Intl. Symp. on Computer Architecture", pp. 39-50, 1998.

[7] R. Hankins et. al, "Scaling and Characterizing Database Workloads: Bridging the Gap between Research and Practice," in Intl. Symp. on Microarchitecture, 2003.

[8] Y. Chou, B. Fahs and S. Abraham, "Microarchitecture Optimizations for Exploiting Memory-Level Parallelism", Intl. Symp. on Computer Architecture", pp. 76, 2004.

[9] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs", IEEE Trans. on Computers, vol. 47, no. 7, pp. 251-248, 1979.

[10] K. Gharachorloo et. al., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", Intl. Symp. on Computer Architecture", pp. 15-26, 1990.

[11] J. Goodman, M. Vernon and P. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors", Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 64-73, April 1989.

[12] M. Dubois, C. Scheurich and F. Briggs, "Memory Access Buffering in Multiprocessors", Intl. Symp. on Computer Architecture, pp. 434-442, June 1986.

[13] R. Rajwar and J. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution", Intl. Symp. on Microarchitecture, pp. 294-305, 2001.

[14] M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Objects", ACM TOPLAS, 15(5):745-770, November 1993.

[15] S. Chaudhry, S. Yip, P. Caprioli and M. Tremblay, "High Performance Throughput Computing", IEEE MICRO Vol. 25 Issue 3, 2005.

[16] J. Dundas and T. Mudge, "Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss", Intl. Conf. on Supercomputing, 1997.

[17] O. Mutlu, J. Stark, C. Wilkerson and Y. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors", Intl. Symp. on High Performance Computer Architecture, 2003.

[18] I. Park, C. Ooi, and V. Vijaykumar, "Reducing Design Complexity of the Load/Store Queue", Intl. Symp. on Microarchitecture, 2003.

[19] A. Gandhi et al, "Scalable Load and Store Processing in Latency Tolerant Processors", Intl. Symp. on Computer Architecture, 2005.

[20] T. Tsuei and W. Yamamoto, "Queuing Simulation Model for Multiprocessor Systems", IEEE Computer, Vol. 36 Issue 2, pp. 58-64, 2003.

[21] K. Gharachorloo, A. Gupta and J. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models", Intl. Conf. on Parallel Processing, pp. 1355-1364, 1991.

[22] L. Spracklen and S. G. Abraham, "Chip Multithreading: Opportunities and Challenges", Intl. Symp. on High-Performance Computer Architecture, pp. 248-252, 2005.

[23] Sun Microsystems. SPARC Architecture Manual V9, 1996.

[24] Book E: Enhanced PowerPC Architecture, Version 1.0, May 2002, Chapter 6.1.6.

[25] R. Bhargava and L. John, "Issues in the Design of Store Buffers in Dynamically Scheduled Processors", Intl. Symp. on Performance Analysis of Systems and Software, pp. 76-87, 2000.

[26] F. Mounes-Toussi and D. Lilja, "Write Buffer Design for Cache-Coherent Shared-Memory Multiprocessors", Intl. Conf. on Computer Design, pp. 506-511, 1995.

[27] N. Jouppi, "Cache Write Policies and Performance", Intl. Symp. on Computer Architecture, pp. 191-201, 1993.

[28] J. Sahuquillo and A. Pont, "Impact of Reducing Miss Write Latencies in Multiprocessors with Two Level Cache", EUROMICRO Conference, pp. 333-336, 1998.

[29] C. Gniady, B. Falsafi, and V. Vijaykumar, "Is SC + ILP = RC?", Intl. Symp. on Computer Architecture, pp. 162 – 171, 1999.

[30] P. Ranganathan, et. al., "Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models", Symp. on Parallel Algorithms and Architectures, pp. 199-210, 1997.

[31] Y. Sohn, N. Jung and S. Maeng, "Request Reordering to Enhance the Performance of Strict Consistency Models", Computer Architecture Letters, 2002.

[32] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking", Intl. Symp. on Computer Architecture, pp. 246-257, 2005.

[33] A. Moshovos, "RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence", Intl. Symp. on Computer Architecture, pp. 234-245, 2005.

[34] D. Sorin et. al., "Analytic Evaluation of Shared-Memory Systems with ILP Processors", Intl. Symp. on Computer Architecture", pp. 380-391, 1998.